Equivalence of Program Dependence Graphs in Refactoring

Lukas Felber

December 20, 2008

Program dependence graphs are widely used in computer science for example when performing optimizations, parallelization and as an internal program representation in integrated development environments. In this paper, I will look into the usefulness of program dependence graphs regarding refactoring.

Refactoring is a very useful technique to improve the internal structure of a program. Many modern integrated development environments provide automated refactoring support because the process of refactoring, when done by hand is time-consuming and error-prone.

This article summarizes three papers which are then used as a basis to develop ideas on the usefulness of program dependence graphs in automated refactoring. These papers all define equivalence theorems for a range of languages which, for some programs, with the use of program dependence graphs, can show that the programs are strongly equivalent when considering a given initial and a resulting final state. This means that two programs can represent the same program behavior even if they are not exactly the same. Since this is often the case for program code before and after refactoring, I examine in this article whether the ideas behind those theorems could be useful in the process of automated refactoring.

It is discussed whether an extension of those equivalence theorems would be possible and necessary, to make them feasible for automated refactorings by guaranteeing that an automatically applied refactoring indeed preserve the functionality of the program.
1 Introduction

Program Dependence Graphs (PDG) are directed graphs which are used to represent a program’s structure. These graphs are utilized in a variety of application areas that include optimization, vectorization and parallelization. Depending on the need of an application, different variations of PDG were introduced. These include [KMC72, Kuc78, KKP+81] by Cuck and [Tow76] by Towle. Here in this article, the PDG defined by Horwitz in [HRB90] will be used as a basis.

The ideas developed in this article are based on the proofs that are given in the three papers that will be introduced in detail in Section 2. The following paragraphs will give a short introduction to these papers.

In On the Adequacy of Program Dependence Graphs for Representing Programs [HPR88] a PDG is defined for a simple sequential programming language that supports neither function calls nor pointers. Then the Equivalence Theorem is defined and proven true. It states that in some programs are still strongly equivalent (but not necessarily exactly the same) when concerning only a given input state and the defined, resulting output state. Used for the proof is the PDG representation of those programs. The papers described in the next two sections extend and redefine the Equivalence Theorem for more complex languages.
The second paper, titled *The Multi-Procedure Equivalence Theorem* [BHR89] can be described as an extension of the *Equivalence Theorem* that covers a language that contains procedure calls. So the *Multi-Procedure Equivalence Theorem* can prove strong equivalence between two programs with the help of graphs that also comprise procedures and calls to procedures.

In the third introduced paper *On the Adequacy of Dependence-Based Representations for Programs with Heaps*, Phil Pfeiffer and Rebeca Parsons Selke introduce another equivalence theorem called the *Pointer-Language Equivalence Theorem* which expands the adequacy of graphs to represent a program’s execution behavior for a programming language that supports heap-allocated storage and reference variables.

In this paper they also introduce a theorem called the *Pointer-Language Slicing Theorem* which proves strong equivalence of a slice of a program and the program itself. Since slicing is not directly the concern of this article, I will not go deeply into this topic.

All these equivalence theorems can show certain programs strongly equivalent. If a theorem proves strong equivalence, then we for sure know the two programs have exactly the same behavior. But these theorems have also limitations. Namely, they cannot give proof that if two programs are not strongly equivalent, then they do also have different program behavior. This means that even if an equivalence theorem says two programs are not strongly equivalent they can still have exactly the same behavior. You can see a sample illustration of this limitation in Section 2.1.3.

Refactoring means to improve the design of existing code without changing the internal functionality of a software system. These steps are often inevitable. Modern integrated development environments such as Eclipse provide a wide variety of support for automated refactoring. This will help a developer to save a lot of time and to avoid many mistakes.

In this paper some ideas are developed on how the process of automated refactoring could be enhanced by providing helpful validation technique that proves the preservation of the functionality of a refactored program with the help of PDGs.

In chapter 2, the three papers under consideration will be introduced in detail by summarizing the papers’ content and by reflecting the meaning and proof of the different equivalence theorems. Section 3 will introduce some examples of refactorings and show how and under which circumstances PGDs and the introduced theorems might help while validating automated refactoring.
1.1 Terms Overview

A short overview about terms that are often used in and necessary to comprehend this paper are given in this section.

PDG
PDG means Program Dependence Graph. This abbreviation is in the first case used when generally talking about a program represented by a graph. The first paper under consideration [HPR88] does not define a special name for the used graphs so the term PDG is also used to refer to those.

SDG
SDG implies the term System Dependence Graph. It describes a combination of several simple PDGs that represent each a program procedure and shows how they are combined using procedure calls. This type of graph is used in the second paper under consideration [BHR89].

SPDG and HPDG
Both Dynamic Scalar Dependence Graphs and Dynamic Heap Dependence Graphs are very similar to the PDG introduced in the first paper under consideration [HPR88]. The significant difference is that in certain cases, some edges might be omitted under certain criteria.

HPDGs are used to represent programs in a programming language that supports pointers and heap allocated storage. SPDG do the same for a language without pointers and heap storage. Both SPDG and HPDG are used in the third paper under consideration [PS91].

Strong Equivalence
The term strong equivalence is used in all of the three introduced papers and refers to two programs that can be considered equals because they have similar PDGs even if the two programs and PDGs are not necessarily exactly the same.

1.2 Program Dependence Graph Introduction

Program Dependence Graphs (PDG) are used in different forms in the following sections. The exact forms are depending on the complexity of the language they represent. Each language will be introduced in the paper’s section.

In Figure 1 you can see the simplest form of a PDG and its corresponding program. A detailed introduction to the corresponding language will follow in Section 2.1.

The following terms are common to all these graph. The descriptions of the edge-types are simplified. They give an idea to what the meaning is but do not contain the complete details of the definition.
1.2.1 Vertices

In a PDG, a vertices represents a program statement. This does not holds however if the statement itself is a statement list. In that case only the statements contained in that list are each represented as vertices. Special cases are statements which itself contains statement lists, e.g. if and while statements. Here the condition of the statement itself is contained in a vertex and the contained statement list’s children each in one as well.

1.2.2 Edges

In a graph, vertices are connected with edges. There are different types of edges that have different meanings. In the simplest graph version, the following kind of edges exist:

control dependence edge This kind of edge characterizes the control flow of a program. In a PDG there exists control dependence edges between a statement that contains a statement list and each statement contained in the list. Elements containing statement lists are entry, while and ifThenElse.

data dependence edge A data dependence edge represents each data- (or value-)flow in a program. This means that for example a vertex that assigns the variable “i” is the source of a data dependence edge pointing to a target vertex ”if i > 0”.

def-order dependence edge A def-order edge is a special kind of data dependence edge. It is an edge between two vertices v1 and v2. For a def-order edge there exists the two conditions (1) that both v1 and v2 assign the same variable and (2) that there are two normal data dependence edges which have v1 and v2 as source and both a third vertex v3 as target. To better understand this, have a look at the def-order dependence edges in Figure 2 or the more complex ones in Figure 1.
2 Equivalence Theorems on PDGs

The three papers introduced here define theorems to show equivalence between different types of programs using various PDG types. In this section you can find a detailed explanation on what the theorems are and how they are proven to be true.

The main aim of the following paper descriptions is to show the theorems’ meaning and not to reflect the overall proof of each theorem in detail.

2.1 On the Adequacy of Program Dependence Graphs for Representing Programs

The paper [HPR88] written by Susan Horwitz, Jan Prins and Thomas Reps is the first paper of the three papers under consideration.

2.1.1 Paper Introduction

The authors concerned themselves with the topic whether a PDG is adequate to represent a program’s execution behavior. Execution behavior is seen as the generated output corresponding to a given input. To show this execution behavior, they introduce the Equivalence Theorem that defines the concept of strong equivalence. They define that two programs are strongly equivalent if they produce the same result with a given input state. The same result can take as (1) that both program diverges or (2) halt in the same final state meaning they produce the same result value set. If, with the help of rules given in the paper, two programs’ PDGs can be shown isomorphic then this proves that the two programs are strongly equivalent.

2.1.2 Language Syntax

The language that is used in this paper is a very simple one that does not supports classes, functions, pointers etc. The following paragraph shows the elements supported by the language as given in [HPR88].
Assign: \( lvalue \times exp \rightarrow stmt \)
While: \( exp \times stmt\_list \rightarrow stmt \)
IfThenElse: \( exp \times stmt\_list \times stmt\_list \rightarrow stmt \)
StmtList: \( stmt \times stmt \times \ldots \times stmt \rightarrow stmt\_list \)
Program: \( stmt\_list \rightarrow program \)

Note that the notation \( a \times b \times \ldots \times c \rightarrow d \) means that an element of type \( d \) consists of the elements \( b \ldots c \). When you look at the IfThenElse element this means that it contains two statement lists (one for the then branch and one for the else branch).

2.1.3 Isomorphism

The term *isomorphism* is used when talking about PDGs or subgraphs of PDGs. If two (sub-)graphs are isomorphic, that means that for each element (including further subgraphs) contained in the primal graph, there exist a correlating element in the second graph. Note that this does not include the same order of the elements defined by a statement list.

**Isomorphism Samples**  Figure 3 shows two PDG and their corresponding programs. The programs are strongly equivalent and the PDG are isomorphic. The isomorphic property is used in the proof of the equivalence theorem.

The following two PDGs (Figure 4) shows two programs that do exactly the same in two different ways. Using the Equivalence Theorem on those two graphs, it says that they are not strongly equivalent. So these graphs are an example for the limitations of the Equivalence Theorem. They demonstrate that the theorem can in certain cases prove two graphs strongly equivalent. But it cannot do the inverse, meaning that two
programs, even if they are claimed not to be *strongly equivalent*, can still be the same when looking at a given input and output state. So actually, the two graphs fulfill the exact conditions of *strong equivalence*.

Figure 4: Both the program calculate the summation from 1 to 10.

The problem which cannot be solved that is shown in Figure 4 is strongly correlated to the *Halting Problem* [Hal]. Solving it would mean solving the *Halting Problem* which in all likelihood is impossible.

### 2.1.4 The Equivalence Theorem and Its Proof

Two programs are defined *strongly equivalent* when their PDGs are *isomorphic*. *Isomorphism* between two PDGs means that they represent the same program behavior meaning when given the same set of input variables they produce the same set of output variables. If two PDGs are *isomorphic* they look very similar. But the statements contained in a statement list can for example be permuted in certain cases.

The proof of the equivalence theorem is given by structural induction. To find out if two PDGs are *isomorphic*, each subgraph of the first PDG is tried to be shown *isomorphic* to the corresponding subgraph of the second PDG. For the language elements *Assign*, *While*, *IfThenElse* and *Program* the proof is simple. First the element itself is compared, then all the incoming and outgoing flow dependence edges are compared. If all of these match the two subgraphs are isomorphic.

Because the elements that are contained in a *StmtList* can be permuted, the proof for isomorphism gets more complicated. To give the proof, a technique called *semantic*
flattening is used. This means that the statements contained in the statement list get translated into another language which consists only of assignment statements and only permits straight-line code. These properties of this second language help to prove the isomorphism of the two statement lists by allowing the gathering of "traces" of the programs execution. These traces then can be compared to each other which shows the equivalence of the two statement lists even if the elements contained in the list do not have the same order.

The original theorem text is stated as following:

THEOREM (EQUIVALENCE THEOREM). If $P$ and $Q$ are programs with isomorphic program dependence graphs, then for any initial state $\sigma$, either both $P$ and $Q$ diverge when initiated on $\sigma$, or both halt with the same final state.

2.2 The Multi-Procedure Equivalence Theorem

The Multi-Procedure Equivalence Theorem [BHR89], written by David Binkley, Susan Horwitz and Thomas Reps is an extension on the first paper [HPR88] that was introduced above. This extension seems very important to me because the enlargement of the scope to a language with procedures pushes the equivalence theorems much nearer to a modern object-oriented programming language and thus make a big step forward when considering the idea of using the introduced theorems in modern software engineering.

2.2.1 Paper Introduction

As can get derived from the paper’s title, the main result of this paper is the introduction of the Multi-Procedure Equivalence Lemma. This theorem states exactly the same as the Equivalence Theorem. But in addition it also covers the topic of procedures and procedure calls. Programs that contains procedures and procedure calls are henceforth referred to as systems.

This paper uses the System Dependence Graph representation introduced in section 2.2.3 to represent those systems. Arguments passed to the procedures are implicitly passed in the call-by-reference manner which can be seen in the sample SDG shown in Figure 5.

Multi-Procedure Equivalence Lemma namely says that two systems are strongly equivalent if both systems, when initiated on the same initial state, also produce the same result (final state) or both diverge.

The Equivalence Theorem is used as a basis to prove the Multi-Procedure Equivalence Theorem. To allow the usage of the Equivalence Theorem, a technique called in-line expansion is used to convert a SDG to a ordinary PDG while preserving the programs semantics.
2.2.2 Language Syntax

The second’s paper language definition can be found here. The first part is the same definition that was given in the first’s paper introduction. The extensions to the first’s paper definition are marked as bold.

Assign: \text{lvalue} \times \text{exp} \rightarrow \text{stmt}
While: \text{exp} \times \text{stmt} \rightarrow \text{stmt}
IfThenElse: \text{exp} \times \text{stmt} \times \text{stmt} \rightarrow \text{stmt}
StmtList: \text{stmt} \times \text{stmt} \times \ldots \times \text{stmt} \rightarrow \text{stmt} \times \text{stmt}
Program: \text{stmt} \rightarrow \text{program}
Arguments: \text{lvalue} \times \text{lvalue} \times \ldots \times \text{lvalue} \rightarrow \text{args}
Procedure: \text{proc} \times \text{args} \times \text{stmt} \rightarrow \text{procedure}
ProcedureCall: \text{proc} \times \text{args} \rightarrow \text{stmt}

As you can see in the last definition line, there is no return statement. Returning values though is possible because procedure calls always are call-by-reference. Since the call-by-reference construct is already supported, an extension of the language and it’s SDG definition with return values is straight forward and would also not hinder the proof of the Multi-Procedure Equivalence Theorem.

2.2.3 System Dependence Graph

A system dependence graph (SDG) is a collection of normal PDGs combined with additional vertices that express the correlation between the calling and the called PDG. A single one of these PDGs represents the main / entry procedure. The other represent procedures that are called by one or several other procedures.

Calling a Procedure

The call of a procedure is represented in a SDG with several vertices. First there is the call vertex that is contained in graph P that calls the procedure as a statement, meaning it is the target of a control flow edge that also originates in P.

For each of the arguments passed to the procedure there exists an assignment vertex, called an actual-in vertex, that contains a variable definition \( x_{in} = \text{AddrOf}(a) \) where \( x \) is the name of the argument in the procedure and \( a \) the name of the variable used in the procedure call. Then, in the graph \( Q \) of the called procedure itself, the value of \( x_{in} \) gets assigned to \( x \) which is interpreted as an implicit dereferencing of the address stored in \( x_{in} \).

For each argument that gets manipulated in \( Q \) there also exists a actual-out vertex that reassigns the new value of an argument \( x \) to the variable \( a \) that was used in the procedure call.

Contained in the SDG is a control flow dependencies from the call vertex to each actual-in and actual-out vertex.
Interprocedural Dependence Edge

In addition to the newly introduced vertices there are several interprocedural dependence edges. The first one that represents the initiation of the procedure call is the call edge which has the call vertex as source and the entry point of the called procedure as target. Then from each actual-in vertex there exists an parameter-in edge to the corresponding initial assignment vertex in the called procedure. Accordingly there exists a parameter-out edge for and from each changed argument in the called procedure to the corresponding actual-out vertex.

Example Graph

In Figure 5 you can see a system and its SDG containing a procedure Add and two calls to this procedure.

![Example Graph](image)

Figure 5: A Sample SDG and its program.

2.2.4 The Multi-Procedure Equivalence Theorem and Its Proof

To use the Equivalence Theorem as a base for the proof of the Multi-Procedure Equivalence Theorem, the whole SDG has to be converted to the simpler PDG graph type that is referred to in the Equivalence Theorem itself. It also has to be proven that the conversion process preserves the functional meaning of the transformed program.

The authors introduce the Expansion Lemma that states that when the content of a procedure gets inlined (called an inline expansion) into a call to the procedure (see
inline method refactoring in [Fow99]) then the original and the newly created system as well as the correlating SDGs are strongly equivalent.

They prove theExpansion Lemma by describing the inlining process stepwise and showing that the semantic of the program stays the same while doing so. Their description of the inlining process also contains the renaming of parameters and local variables of the inlined procedure if there are any naming conflicts involved.

TheExpansion Lemma is already sufficient to prove theMulti-Procedure Equivalence Theorem in the case when the considered system contains no recursive method calls. In the case that the system contains recursive method calls, theFlattening Lemma is needed to prove the strong equivalence of two systems.

In the proof of theFlattening Lemma a construct called an invocation tree is introduced. The invocation tree is a graph containing a vertex for each particular procedure call in a system’s execution with a given initial state. A system that terminates has an invocation tree of finite size. It is shown that with the help of theExpansion Lemma it is possible to reduce the invocation tree to a size of 1, which means that there are no recursive procedure calls anymore and the SDG thus became an PDG. At this point we again can use theEquivalence Theorem to finish the proof of the strong equivalence of two systems.

The original theorem text is stated as following:

**THEOREM** (MULTI-PROCEDURE EQUIVALENCE THEOREM). If S and T are systems with isomorphic system dependence graphs then S and T are strongly equivalent.

### 2.3 On the Adequacy of Dependence-Based Representations for Programs with Heaps

Like the last introduced paper [BHR89], the paper discussed in this section, On the Adequacy of Dependence-Based Representations for Programs with Heaps [PS91], written by Phil Pfeiffer and Rebecca Parsons Selke, is an extension of the first paper under consideration [HPR88]. Also this extension is an important step into the direction of a modern programming language considering strong equivalence of PDG since references and heaps are used in practically every modern language.

I received a rather negative impression of this paper because in many places it is not clearly phrased. Often terms and expression are used while it is not clear what their meaning is. There are also no references made on the origin of these unclear terms.

#### 2.3.1 Paper Introduction

The extension is again concerned with strong equivalence of PDGs. The aim of the paper is to introduce two equivalence theorems so that certain PDGs for a programming language with support for heap-allocated storage can be proven isomorphic. The elements which represent heap-allocated storage in the considered language can be compared to
the \textit{cons-cells} in the programming language \textit{Lisp} [Lis]. Put simply, a \textit{cons-cell} is a pair of variables.

A variable can either contain a value or be a reference to a \textit{cons-cell}. A part of a \textit{cons-cell}, since the part itself is again a variable, can be a reference to another \textit{cons-cell}. Like this it is simple to create lists and trees.

The main part of this section will focus on the \textit{Pointer-Language Equivalence Theorem} because this theorem is what is important to the ideas developed in Section 3.

The procedure to prove the \textit{Pointer-Language Equivalence Theorem} is very similar to the one used to prove the \textit{Multi-Procedure Equivalence Theorem} in Section 2.2. A program and the representation of the program’s used memory are reduced to a language and a memory representation that do not contain pointers. We will call the source language, which’s syntax is described in Section 2.3.2, \textit{H} and the target language \textit{S}. \textit{H} and \textit{S} correspond to the words "heap" and "scalar" which describe the characteristic of its language.

The second theorem, the \textit{Pointer-Language Slicing Theorem}, concerns itself with the prove that a program’s slices are adequate to represent that slice’s execution behavior when considering the same initial state for both the program and the slice. The preservation of the execution behavior means that both produce the same output when considering only the output statements contained in the slice.

\subsection*{2.3.2 Language Syntax}

This paper’s language is an extension of the language introduce in the first paper under consideration. The extension contained the second paper under consideration, procedures and procedure calls, however is not included in this one. This language supports besides basic language constructs also references to heap allocated memory.

\begin{verbatim}
Program:   → Stmt_list
Stmt_list  → Stmt { ; Stmt } *
Stmt       → while Cond do Stmt_list od
Stmt       → if Cond then Stmt_list else Stmt_list fi
Stmt       → IdExp := Exp
Cond       → isAtom simpleExp | isNil simpleExp | not Cond
Exp        → SimpleExp | Exp::Exp
SimpleExp  → ATOM | IdExp
IdExp      → IDENT{ . sel } *
Sel        → hd | tl
\end{verbatim}

\textit{ATOM} is a set of primitive objects - e.g. integers. \textit{IDENT} is a set of lower-case alphanumeric identifier names. Members of \textit{IdExp} are called \textit{identifier expressions}. \textit{hd} and \textit{tl} stands for head and tail, respectively.
The term \( \text{Exp}::\text{Exp} \) represents the two parts of a \textit{cons-cell} as it is known e.g. in the programming language \textit{Lisp} \[\text{Lis}\]. In \textit{Lisp} the two parts are not called \textit{hd} and \textit{tl} but \textit{car} and \textit{cdr}.

### 2.3.3 SPDG and HPDG

As already mentioned in the first Section, the \textit{SPDG} and \textit{HPDG} graph type do both look very similar to a normal PDG as described in 2.1. The only relevant differences are that \textit{data dependence edges} are called \textit{output dependence edges} and that a \textit{transitive output edge} is introduced. For consistency reason, I will still call the former \textit{data dependence edges}.

A \textit{transitive output dependence edge} is meant to improve the visual representation of a graph because it makes it possible to omit a row of data dependence edges.

The authors refer to those normal PDGs including \textit{transitive output dependence edge} as \textit{static PDG}.

### 2.3.4 HPDG

A \textit{dynamic heap PDG} is a \textit{static PDG} which omits all the \textit{data flow} edges which’s target vertex will never be evaluated because the evaluation is not possible based on a predicate vertex which’s condition is uniformly false. This is why the graph is called ”dynamic”.

A \textit{HPDG} represents a program in the languages \( H \) which’s syntax was given above. This means the language with support for heap-allocated storage and thus the one before the program reduction.

\textbf{SPDG}  Also a \textit{dynamic scalar PDG} is a \textit{static PDG}. Because both \textit{HPDG} and \textit{SPDG} are \textit{dynamic}, also the \textit{SPDG} omits the edges that are described in the \textit{HPDG} section.

\textit{SPDG}’s are used to represent programs in the language \( S \) that does not support heap-allocated storage, meaning it represents a given program after its reduction.

### 2.3.5 Pointer-Language Equivalence Theorem and Its Proof

To prove that two graphs in \( H \) are \textit{isomorph}, it is necessary to reduce both the program and states of the program to the target language \( S \). This means the two resulting programs in \( S \) respectively their \textit{SPDG} then can be proven \textit{isomorph}. This, by the way, is not achieved through the usage of the \textit{Equivalence Theorem} proven by S. Horwitz in \[\text{HPR88}\] but with a graph-rewriting semantic for PDGs described in \[\text{PS89}\].

In the theorem’s proof, the reduction can only be used when it is proven, that the program \( h \) in language \( H \) before the reduction and the resulting program \( s \) in language \( S \) after the reduction represent equivalent execution behavior. This proof is given by showing that both \( h \) and \( s \), initiated on an initial program state, produce and equivalent sequence of states during program execution. So the sequences of states generated by \( h \) and \( s \) needs to be compared to each other.
Because, in the reduction from $h$ to $s$, one statement in $H$ might reduce to $n$ statements in $s$, the comparison between program states is always done after exactly one program step in $h$ and after the corresponding $n$ steps in program $s$.

Understandably, the memory representation of a program state in language $H$ looks different from the one in $S$. So to finish the proof, there is also a mechanism given which converts a program state in $H$ to as corresponding state in $S$. In this mechanism, a cons-cell is converted into several normal variables. If it contains two values then two variables will be enough to convert this cell. References to cons-cells are translated into several variables that represents the reference itself and also the content of the cell.

**THEOREM** (POINTER LANGUAGE EQUIVALENCE THEOREM). Let $P$ and $Q$ be programs the have isomorphic HPDGs, $H_P$ and $H_Q$. If $P$ terminates successfully on $\sigma$, then (1) $Q$ terminates successfully on $\sigma$, (2) $P$ and $Q$ compute equivalent sequences of values at corresponding program points, and (3) $P(\sigma) \approx_H Q(\sigma)$.

3 Usefulness of the Graph Theorems in Refactoring

The goal of refactoring is to change a program's structure while preserving its functionality. This means that a program transformation takes place while the semantics of the program in a general view stays exactly the same. When an application developer refactors manually, it is his own responsibility to see that he does not break the program's functionality.

Considering the theorems of the described papers, this is a very interesting operation because when looking at the program code's structure before and after the refactoring, the code's functionality stays equivalent. This would mean that it might be possible and also feasible to create a PDG with the original source code and one with the refactored source code. Then, with an appropriate extensions of the theorems now known to us, it might be possible to validate that the two programs are strongly equivalent by showing that their PDGs are isomorphic.

Like this there would be a way to guarantee that the functionality of an automated refactoring was preserved meaning a automated refactoring developer would not have to develop a validating concept based on a specific refactoring's functionality. Developing such a concept without this help for each refactoring right now is time consuming and error-prone. Here the help of an equivalence theorem could save a lot of time.

3.1 Testing of Refactoring

Lets assume that there exists an implementation of an equivalence theorem for a modern object-oriented programming language. Then a programmer that concerns himself with the development of refactorings for this language could use the equivalence theorem implementation to create unit tests that can always be predicted if a refactoring worked as supposed to because the equivalence theorem guarantees the behavioral equivalence of
the program code before and after the refactoring when the refactoring was implemented correctly. A developer could even easier use the *test first* programming principle since the theorem’s implementation is given.

### 3.2 Narrowing Down on Example Refactorings

Here, some refactorings are described that might profit from an implementation of an *equivalence theorem*. For detailed descriptions see the book *Refactoring* of Martin Fowler [Fow99]. There might also be other refactorings, besides the ones mentioned here, that can profit form an *equivalence theorem*.

#### 3.2.1 Inline Method

In the explanation of the *Mulit-Procedure Equivalence Theorem*, the *inline expansion* of a system’s procedures is explained. The system before and after the expansion is proven to have *isomorphic* PDG. Since this procedure correlates directly with the *Inline Method* refactoring, the *Mulit-Procedure Equivalence Theorem* should be sufficient to prove the needed *strong equivalence*.

#### 3.2.2 Extract Method

Being exactly the reverse operation of the *Inline Method* refactoring, the *Extract Method* refactoring can profit from the *Mulit-Procedure Equivalence Theorem* in exactly the same way.

#### 3.2.3 Rename

All the different introduced *equivalence theorems* do not directly help in the case of a *Rename* refactoring because the equivalence of variables is only given when also the variable name is equals. The way the *equivalence theorems* are defined, it should be easy to have them also support different variable names. Instead of comparing the exact name of variables used in a vertex, one could just compare the variables type and its position of use. As a help a name mapping table could be created so that for example the first variable of type *int* used in a procedure could be proven *strongly equivalent* to the corresponding first variable in an other procedure even while having a different name.

### 3.3 Finding equivalent code sequences

A nice addition to an *Extract Method* refactoring is the automated finding of equivalent code blocks that can be replaced with a call to the method that gets extracted. In a normal case *equivalent* means the same blocks with different variable names. Almost all implementation of todays automated refactorings provide this functionality. With an implementation of an *equivalence theorem* for a given language, it might be possible to find more than just equal blocks with different variable names. An example for what could be found as equals when extracting a method is shown in Figure 3.
### 3.4 Drawbacks

In this section I give some points that might be a hindrance when trying to develop an equivalence theorem for a modern programming language. Here I assume that this language supports all the features that might be problematic. This includes for example class inheritance, reflection, direct memory addressing and pointer manipulation.

**Inheritance**  When a class $A$ of an object oriented programming language derives methods from a base class $B$, this means that the inheriting class is given the possibility to override a method $M$ of class $B$ which results in the method $M'$.

Now while trying to build a PDG of a method that receives an argument of class $A$, which means that the argument could also be of type $B$, there will be problems when there exists method calls to the method $M$. It cannot be decided if the method that will actually be called is the method $M$ or $M'$.

The only workaround for this problem that I can think of is to somehow include the worst case, meaning both a call to $M$ and $M'$, in the PDG. In a complex construct of inheriting classes a PDG will get messed up and its usefulness to show isomorphism might be lowered dramatically.

**Input and Output**  In the languages contained in the introduced papers, there exists only the possibility to make input and output at the begin and the end of the program. This might be a hindrance when trying to declare an equivalence theorem for a modern programming language where input and output operations might happen all the time.

**Direct Address Manipulation**  In some languages, like C/C++, there exists the possibility to manipulate memory addresses by manipulating the values of pointers. For an equivalence theorem such operations are a huge problem because whenever a pointer is manipulated and later on dereferenced there exists no way to predict to which variable it points at the current program point. This causes the process of building a useful PDG to become almost impossible.

Following you can see a program code that illustrates the problem:

```c
int main( int argc , const char* argv[] )
{
    int arr[10];
    int* p;
    p = &arr[2];  // 'p' points to the third element of 'arr'
    p += sizeof(int);  // Increment 'p'
    p += atoi(argv[1]);  // reads program argument, converts it to int
                         // and increment 'p' with that value
    *p = 15;  // Assigns 15 to 'p's target
}
```

After the execution of line 6 it can still be determined to what location $p$ points to (namely the fourth element of $arr$), whereas after the end of the execution of the program it is unknown which of the elements of the integer array $arr$ contains the value 15.
There are two even worse cases. Firstly the pointer $p$ may point to a memory location that is not in the range of the array at all (argv[1] might have been "-137"). Secondly, what is the worst scenario, there might be a segmentation error because of an illegal memory address operation.

### 3.5 Feasibility

In this section you find my personal view about the practicability of the developed ideas in Section 3. When considering the possible drawbacks listed above I do not think that a proof of an equivalence theorem for a modern programming language like Java, C++ or C# can be given. Apart from that, if someone tried to implement such an equivalence theorem there will most likely be serious performance problems when creating SDGs for bigger software systems and even more serious ones when trying to prove two SDGs isomorph. All these reduction steps that are described in the three papers under consideration and some transformation rules given to solve minor problems will, when implemented, not scale at all.

When looking at all the possible drawbacks it seems clear that there exists no possibility to expand an equivalence theorem so it covers every possible refactoring. This was already clear in the beginning because this would mean to successfully solve a problem of the complexity of the halting problem. Because of this the following section will concern with the usefulness of equivalence theorems for some chosen refactorings.

**Strong Equivalence for Particular Refactorings**

When looking at a particular situation of a certain refactoring, lets for now think of rename and extract method, I am confident that theorems could be proven which shows strong equivalence of program code before and after refactoring. For this it would not be necessary to build a SDG of a whole software system. A little part of it would be enough. I could also think that in many cases an SDG would not have to include whole called procedures in its structure because as long as the called procedure is not changed and the used arguments, their types and their order are the same, then equivalence is assured anyway.

I am no sure if the process of building an PDG/SDG in such a scenario is even necessary. To run a program in any programming language I now of, an Abstract Syntax Tree (AST) is created. As far as I’m concerned these ASTs should be sufficient to build the theorems on for a particular refactoring.
References


